Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2013-016 July 17, 2013

Coded Emulation of Shared Atomic Memory for Message Passing Architectures

Viveck R. Cadambe, Nancy Lynch, Muriel Medard, and Peter Musial

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Coded Emulation of Shared Atomic Memory for Message Passing Architectures

Viveck R. Cadambe, Nancy Lynch, Muriel Médard, Peter Musial*

Abstract. This paper considers the communication and storage costs of emulating atomic (linearizable) read/write shared memory in distributed message-passing systems. We analyze the costs of previously-proposed algorithms by Attiya, Bar-Noy, and Dolev (the *ABD algorithm*) and by Fan and Lynch (the *LDR algorithm*), and develop new coding-based algorithms that significantly reduce these costs. The paper contains three main contributions:
(1) We present a new shared-memory algorithm that we call *CAS*, for *Coded Atomic Storage*. This algorithm uses *erasure coding* methods.
(2) In a storage system with N servers that is resilient to f server failures, we show that the communication costs for the ABD and LDR algorithms, measured in terms of number of object values, are both at least $f + 1$, whereas the communication cost for CAS is $\frac{N}{N-2f}$.
(3) We also explicitly quantify the storage costs of the ABD, LDR, and CAS algorithms. The storage cost of the ABD algorithm, measured in terms of number of object values, is N, whereas the storage costs of the LDR and CAS algorithms are both unbounded. We present a modification of the CAS algorithm based on the idea of garbage collection. The modified version of CAS has a storage cost of $(\delta + 1)\frac{N}{N-2f}$, where δ in an upper bound on the number of operations that are concurrent with a read operation. Thus, if δ is sufficiently small, the storage cost of CAS is lower than those of both the ABD and LDR algorithms.

1 Introduction

Since the late 1970s, emulation of shared-memory systems in distributed message-passing environments has been an active area of research [9, 20, 4, 16, 10, 3, 8]. The most typical approach to building redundancy for distributed systems in the context of shared memory emulation is *replication*. Replication techniques in message-passing systems were pioneered by Gifford [9] and Thomas [20]. In [4], Attiya, Bar-Noy, and Dolev presented an algorithm for emulating shared memory that achieves atomic consistency [12, 11]. Their algorithm allows only a single node to act as a writer; in this paper we consider a simple multi-writer generalization of their algorithm which we call the *ABD* algorithm[i]. Their algorithm uses a quorum-based replication scheme [21], combined with read and write protocols to ensure that the emulated object is atomic [12] (linearizable [11]), and to ensure liveness, specifically, that each operation terminates provided that at most $\lceil \frac{N-1}{2} \rceil$ server nodes fail. A critical step in ensuring atomicity in the ABD algorithm is the *propagate* phase of the read protocol, where the readers write back the value they read to

* Viveck R. Cadambe and Muriel Médard are with the Research Laboratory of Electronics (RLE), MIT, Cambridge, USA [viveck, medard]@mit.edu. Nancy Lynch is with Computer Science and Artificial Intelligence Laboratory (CSAIL), MIT, Cambridge, USA lynch@theory.lcs.mit.edu. Peter Musial is the Advanced Storage Division of EMC^2, Cambridge, USA peter.musial@emc.com

[i] The algorithm of Attiya, Bar-Noy and Dolev [4] did not distinguish between client and server nodes as we do in our paper.

a subset of the nodes in the storage system. Since the read and write protocols require multiple communication phases where entire replicas are sent, this algorithm has a high communication cost.

To the best of our knowledge the only work that has previously studied *efficient* emulation of atomic shared memory is by Lynch and Fan [7]. In [7], the authors introduced a directory-based replication system that, like [4], emulates atomic shared memory in the message-passing model; however, unlike [4], the readers are required to write only some *metadata* information to the directory, rather than the value read (thus making them *selfish*). In applications where the data being replicated is much larger than the metadata, the LDR algorithm is less costly than ABD in terms of communication costs.

In this paper, we investigate practical aspects of emulating shared memory in distributed networks. Specifically, we focus on the communication and storage costs of emulating atomic read/write shared memory in distributed message-passing systems. The unique aspect of our work is the merging of *quorum-based techniques* with *erasure coding techniques.*

Erasure coding is a generalization of replication that is well known in the context of classical storage systems [13, 19, 6]. Specifically, in erasure coding, each server does not store the value in its entirety, but only a part of the value called a *coded element*. In the classical coding theory framework, this approach is well known to lead to smaller storage costs[ii] as compared to replication (see Sec. 4). However, classical erasure coding theory does not consider issues of consistency; in fact, the most commonly studied models in the coding literature focus on storing a single version of each data object. A natural question is whether the cost benefits of coding that appear in the context of storing a single version, translate to our setting of storing multiple versions consistently.

The major challenges in integrating erasure coding into quorum-based data management lie in the handling of writer failures and in ensuring termination of read operations. In ABD and similar algorithms, writer failures are not an issue since a writer always sends an entire value to any server, hence readers who observe a trace of a failed write can help complete the write operation. However, this is not the case when coding is used since any single server will contain only some part of the value being written, i.e., a coded element (see Sec. 4). While the benefits of erasure codes have been well demonstrated for distributed data storage systems (see [1, 5, 17, 6]), their applicability in the context of emulation of a coherent shared memory has scarcely been studied. Among the few previous works that are relevant in this context, perhaps the most interesting is [2], where erasure coding is used to devise an atomic shared-memory system. However, the technique of [2] does not ensure termination in the presence of client failures since read operations must wait for ongoing write operations to terminate. In particular, while the system of [2] does support transient server failures, the failure of a single writer can cause a future read operation to be blocked forever.

In this paper, we present algorithms that use erasure coding to emulate atomic shared memory in asynchronous message-passing systems. These algorithms maintain the desirable properties of shared memory systems while yielding significant improvements over traditional techniques in terms of communication and storage costs.

[ii] In much of the literature that applies erasure coding to storage systems, the message passing system is ignored in the modeling and therefore, communication costs are not relevant.

	Read Communication Cost	Write Communication Cost	Storage Cost
ABD	$2N$	N	N
LDR	$2f+1$	$f+1$	∞
CAS	$\frac{N}{N-2f}$	$\frac{N}{N-2f}$	∞
CASGC	$\frac{N}{N-2f}$	$\frac{N}{N-2f}$	$(\delta+1)\times\frac{N}{N-2f}$

Table 1. Worst-case communication and storage costs for the ABD, LDR, CAS and CASGC algorithms (see Theorems 3.1, 3.2, 5.10, 6.1). The parameter δ represents an upper bound on the number of operations that are concurrent with a read operation. The costs are measured in terms of the number of object values (see Sec. 2 for more precise definitions.)

Contributions. As in ABD and LDR, we consider a static distributed message-passing setting where the universe of nodes is fixed and known, and nodes communicate using a reliable message-passing network. We assume that client and/or server nodes can fail. We define our system model, and our communication and storage cost measures in Sec. 2. We provide brief descriptions of the ABD and LDR algorithms and analyze their communication and storage costs in Sec. 3.

Our main contribution is the *Coded Atomic Storage* (CAS) algorithm presented in Sec. 5, which is a lower cost alternative to ABD and LDR. Since CAS uses erasure coding, we present a brief introduction of this technique in Sec. 4. For a storage system with N nodes, CAS ensures liveness provided that the number of *server* failures is at most f and ensures atomicity regardless of the number of (client or server) failures. In Sec. 5, we also analyze the communication cost of CAS.

The communication and storage costs of the ABD, LDR, and CAS algorithms are depicted in Tab. 1. Specifically, in a storage system with N servers that is resilient to f server node failures, the communication costs of CAS are smaller than the communication costs of both ABD and LDR if $\frac{N}{N-2f} < f+1$, i.e., if $N > 2f+2$. The storage cost of the ABD algorithm is N, whereas the storage costs of the LDR and CAS algorithms are unbounded. The reason for this difference is that in ABD, each server stores the value associated with the latest version of the data object it receives, whereas in the LDR and CAS algorithms, each server stores the values or coded elements associated with all the versions of the data object it receives. In executions where an unbounded number of write operations occur, the servers may store values or coded elements corresponding to an unbounded number of object values, thus resulting in unbounded storage costs. In Sec. 6, we present a variant of the CAS algorithm, called the CAS with Garbage Collection (CASGC) algorithm, which has a bounded storage cost; specifically, the CASGC algorithm has a storage cost of $(\delta+1)\frac{N}{N-2f}$, where δ represents an upper bound on the number of operations that are concurrent with a read operation. The CASGC algorithm achieves a bounded storage cost by *garbage collection*, i.e., discarding values associated with sufficiently old versions. We argue in Sec. 6 that if the number of operations that are concurrent with a read is bounded, such garbage collection can be done without loss of atomicity or liveness.

We finish by discussing some open questions and areas of future research in Sec. 7.

2 System Model

Deployment setting. In this work, we assume a *static asynchronous deployment setting* where all the nodes and the network connections are known a priori and the only sources of dynamic behavior are node stop-failures (or simply, failures) and processing

and communication delays. We consider a message-passing setting where nodes communicate via point-to-point reliable channels.

We assume a universe of nodes that is the union of *server* and *client* nodes, where the client nodes are *reader* or *writer* nodes. $\mathcal{N}$ represents the set of server nodes; N denotes the cardinality of $\mathcal{N}$. We assume that server and client nodes can fail (stop execution) at any point. We assume that the number of server node failures is at most f. There is no bound on the number of client failures.

Shared memory emulation. We consider algorithms that emulate multi-writer, multi-reader (MWMR) read/write atomic shared memory using our deployment platform. We assume that read clients receive read requests (invocations) from some local external source, and respond with object values. Write clients receive write requests and respond with acknowledgments. The requests follow a "handshake" discipline, where a new invocation at a client waits for a response to the preceding invocation at the same client. We require that the overall external behavior of the algorithm corresponds to atomic memory. For simplicity, in this paper we consider a shared-memory system that consists of just a single object.

As in [4] we represent each version of the data object as a $(tag, value)$ pair. When a write client processes a write request, it assigns a *tag* to the request. We assume that the tag is an element of a totally ordered set $\mathcal{T}$ that has a minimum element t_0. The tag of a write request serves as a unique identifier for that request, and the tags associated with successive write requests at a particular write client increase monotonically. Different algorithms use different kinds of tags. We assume that $value$ is a member of a finite set $\mathcal{V}$ that represents the set of values that the data object can take on; this implies that $value$ can be represented by $\log_2 |\mathcal{V}|$ bits[iii]. We assume that all servers are initialized with a default initial state.

Requirements. The key correctness requirement on the targeted shared memory service is *atomicity.* A shared atomic object is one that supports concurrent access by multiple clients and where the observed global external behaviors "look like" the object is being accessed sequentially. Another requirement is *liveness*, by which we mean here that an operation of a non-failed client is guaranteed to terminate provided that the number of server failures is at most f, and irrespective of the failures of other clients[iv].

Communication cost. Informally speaking, the communication cost is the number of bits transferred over the point-to-point links in the message-passing system. For a message that can take any value in some finite set $\mathcal{M}$, we measure the cost of the message as $\log_2 |\mathcal{M}|$ bits. In this paper, we aim to separate the cost of communicating a value of the data object from the cost of communicating the tags and other metadata. For this purpose, we assume that each message is a triple[v] (t, w, d) where $t \in \mathcal{T}$ is a tag, $w \in \mathcal{W}$ is the (only) component of the triple that depends on the value associated with the tag t, and $d \in \mathcal{D}$ is any additional metadata that is independent of the value. Here, $\mathcal{W}$ is a finite set of values that the second component of the message can take on, depending on the value

[iii] Strictly speaking, we need $\lceil \log_2 |\mathcal{V}| \rceil$ bits since the number of bits has to be an integer. We ignore this rounding error in this paper.

[iv] We will assume that $N > 2f$, since atomicity cannot be guaranteed if $N \leq 2f$ [15].

[v] It is possible for some messages to have some of the fields empty. For example, if the message carries only metadata, then $\mathcal{W}$ is the empty set. In such a case, we will simply omit the data field. Messages where the metadata is missing are also handled similarly.

of the data object. $\mathcal{D}$ is a finite set that represents all the possible metadata elements for the message. These sets are assumed to be known a priori to the sender and recipient of the message. In this paper, we make the approximation: $\log_2 |\mathcal{M}| \approx \log_2 |\mathcal{W}|$, that is, the costs of communicating the tags and the metadata are negligible as compared to the cost of communicating the data object values, $\log_2 |\mathcal{W}|$. We assume that every message is sent on behalf of some read operation or write operation. We next define the read and write communication costs of an algorithm.

For a given shared memory algorithm, consider an execution α of the system. Consider a write operation in α. The communication cost of this write operation is the sum of the communication costs of all the messages sent over the point-to-point links on behalf of the operation. The write communication cost of the execution α is the supremum of the costs of all the write operations in α. The write communication cost of the algorithm is the supremum of the write communication costs taken over all executions of the algorithm. The read communication cost of an algorithm is defined similarly.

Storage cost. Informally speaking, at any point of an execution of an algorithm, the *storage cost* is the total number of bits stored by the servers. Similarly to how we measured the communication costs, in this paper, we isolate the costs of storing the data object from the cost of storing the tags and the metadata. Specifically, we assume that a server node stores a set of triples with each triple of the form (t, w, d), where $t \in \mathcal{T}$, w depends on the value of the data object associated with tag t, and d represents any additional metadata that is independent of the values stored. We neglect the cost of storing the tags and the metadata; so the cost of storing the triple (t, w, d) is measured as $\log_2 |\mathcal{W}|$ bits. The storage cost of a server is the sum of the storage costs of all the triples stored at the server. For a given shared memory algorithm, consider an execution α. The total storage cost at a particular point of α is the sum of the storage costs of all the servers at that point. The total storage cost of the execution α is the supremum of the storage costs over all points of the execution. The total storage cost of an algorithm is the supremum of the total storage costs over all executions of the algorithm[vi].

3 The ABD and LDR Algorithms

As baselines for our work we use the MWMR versions of the ABD and LDR algorithms [4, 7]. Because of space constraints, here, we will only describe these algorithms informally, in the context of evaluation of their communication and storage costs. The costs of these algorithms are stated in Theorems 3.1 and 3.2. We provide a complete description of ABD and LDR and proofs of Theorems 3.1 and 3.2 in Appendix A. In the ABD algorithm, the write protocol involves communication of a message that includes the value of the data object to each of the N servers, and therefore incurs a write communication cost of $N \log_2 |\mathcal{V}|$ bits. The read communication cost is bigger because it involves communication of the value of the data object in two of its phases, known as the *get* phase and the *put* phase (see Fig. 3 in Appendix A). In the *get* phase, each of the N servers send a message containing a value to the reader, and the communication cost incurred is $N \log_2 |\mathcal{V}|$ bits. In the *put* phase, the reader sends a message involving the value of its data object to each of the N servers, and the communication cost incurred is $N \log_2 |\mathcal{V}|$

[vi] The total storage costs of some of the algorithms in this paper are unbounded. For such algorithms, we measure the storage cost of the algorithm for a class of executions as the supremum of the total storage costs over all executions belonging to that class (in particular, see Sec. 6)

bits. The read communication cost of the ABD algorithm is therefore $2N \log_2 |\mathcal{V}|$ bits. The storage cost of the ABD algorithm is $N \log_2 |\mathcal{V}|$ bits since each server stores exactly one value of the data object.

Theorem 3.1. *The write and read communication costs of ABD are respectively equal to $N \log |\mathcal{V}|$ and $2N \log |\mathcal{V}|$ bits. The storage cost is equal to $N \log_2 |\mathcal{V}|$ bits.*

The LDR algorithm divides its servers into *directory servers* that store metadata, and *replica servers* that store object values. The write protocol of LDR involves the sending of object values to $2f + 1$ replica servers. The read protocol is less taxing since in the worst-case, it involves retrieving the data object values from $f + 1$ replica servers. We state the communication costs of LDR next (for formal proof, see Appendix A.)

Theorem 3.2. *In LDR, the write communication cost is $(2f + 1) \log_2 |\mathcal{V}|$ bits, and the read communication cost is $(f + 1) \log_2 |\mathcal{V}|$ bits.*

In the LDR algorithm, each replica server stores every version of the data object it receives[vii]. Therefore, the (worst-case) storage cost of the LDR algorithm is unbounded. We revisit the issue of storage cost of the LDR algorithm in Sec. 6.

4 Erasure Coding - Background

Erasure coding is a generalization of replication that has been widely studied for purposes of failure-tolerance in storage systems (see [13, 19, 17, 6]). Erasure codes per se do not address the issue of maintaining consistency in distributed storage systems since the classical erasure coding models assume that only a single version of the data object is being stored. In this paper, we use erasure coding as an ancillary technique in providing a shared-memory system that is consistent (specifically atomic) across different versions of the data. Here we give an overview of the relevant definitions and results from classical coding theory [19].

The key idea of erasure coding involves splitting the data into several *coded elements*, each of which is then stored at a different server node. As long as a sufficient number of coded elements can be accessed, the original data can be recovered. Informally speaking, given two positive integers m, k, $k < m$, *an (m, k) Maximum Distance Separable (MDS) code maps a k-length vector to an m-length vector, where the original (input) k-length vector can be recovered from any k coordinates of the output m-length vector.* This implies that an (m, k) code, when used to store a k-length vector on m server nodes - each server node storing one of the m coordinates of the output - can tolerate $(m - k)$ node failures (erasures[viii]) in the absence of any consistency requirements (for example, see [1]). We proceed to define the notion of an MDS code formally.

Given an arbitrary finite set $\mathcal{A}$ and any set $S \subseteq \{1, 2, \ldots, m\}$, let π_S denote the *natural projection mapping* from $\mathcal{A}^m$ onto the coordinates corresponding to S, i.e., denoting $S = \{s_1, s_2, \ldots, s_{|S|}\}$, where $s_1 < s_2 \ldots < s_{|S|}$, the function $\pi_S : \mathcal{A}^m \rightarrow \mathcal{A}^{|S|}$ is defined as $\pi_S (x_1, x_2, \ldots, x_m) = (x_{s_1}, x_{s_2}, \ldots, x_{s_{|S|}})$.

Definition 4.1 ((m, k) Maximum Distance Separable (MDS) code). *Let $\mathcal{A}$ denote any finite set. Given positive integers k, m such that $k < m$, an (m, k) code over $\mathcal{A}$ is*

[vii] This is unlike ABD where the servers store only the latest version of the data object received.

[viii] In information and coding theory literature, an *erasure* is an abstraction that corresponds to a node failure; hence the term *erasure coding*.

a map $\Phi : \mathcal{A}^k \rightarrow \mathcal{A}^m$. An (m, k) code Φ over $\mathcal{A}$ is said to be Maximum Distance Separable *(MDS) if, for every $S \subseteq \{1, 2, \ldots, m\}$ where $|S| = k$, there exists a function $\Phi_S^{-1} : \mathcal{A}^k \rightarrow \mathcal{A}^k$ such that: $\Phi_S^{-1}(\pi_S(\Phi(\mathbf{x})) = \mathbf{x}$ for every $\mathbf{x} \in \mathcal{A}^k$, where π_S is the natural projection mapping.*

We refer to each of the m coordinates of the output of an (m, k) code Φ as a *coded element*. Put simply, an (m, k) MDS code Φ is one where the input to Φ is obtainable from any k coded elements. Classical m-way replication, where the input value is repeated m times, is in fact an $(m, 1)$ MDS code, since the input can be obtained from any single coded element (copy). Another example is the *single parity code*: an $(m, m-1)$ MDS code over $\mathcal{A} = \{0, 1\}$ which maps the $(m-1)$-bit vector $x_1, x_2, \ldots, x_{m-1}$ to the m-bit vector $x_1, x_2, \ldots, x_{m-1}, x_1 \oplus x_2 \oplus \ldots \oplus x_{m-1}$.

Erasure Coding for single-version data storage. In the classical coding-theoretic model, a single version of a data object is stored over N servers. Here we review how an MDS code can be used in this setting.

Consider a single version of the data object with value $v \in \mathcal{V}$. Suppose that we want to store value v among the N servers using an (N, k) MDS code. For simplicity, we will assume here that $\mathcal{V} = \mathcal{W}^k$ for some finite set $\mathcal{W}$ and that an (N, k) MDS code $\Phi : \mathcal{W}^k \rightarrow \mathcal{W}^N$ exists over $\mathcal{W}$ (see Appendix B for a discussion). Then, the value of the data object v can be used as an input to Φ to get N coded elements over $\mathcal{W}$; each of the N servers, respectively, stores one of these coded elements. Since each coded element belongs to the set $\mathcal{W}$, whose cardinality satisfies $|\mathcal{W}| = |\mathcal{V}|^{1/k} = 2^{\frac{\log_2 |\mathcal{V}|}{k}}$, *each coded element can be represented as a $\frac{\log_2 |\mathcal{V}|}{k}$ bit-vector, i.e., the number of bits in each coded element is a fraction $\frac{1}{k}$ of the number of bits in the original data object*. Therefore, the total storage cost of storing the coded elements over N servers is $\frac{N}{k}$ times the number of bits in the original data object. The ratio $\frac{N}{k}$ - also known as the *redundancy factor* of the code - represents the storage cost overhead in the classical erasure coding model. Much literature in classical coding theory is devoted to the study of making the redundancy factor as small as possible (see [19, 13] and discussion in Appendix B).

The redundancy factor of the code turns out to be relevant even in our context of storing multiple versions of the data object in a shared-memory system. The CAS and CASGC algorithms presented in Sec. 5 and 6 both use an (N, k) MDS code Φ to store values of the data object. It turns out that a smaller redundancy factor of the code translates to smaller communication and storage costs (see Theorems 5.10, Theorem 6.1).

5 Coded Atomic Storage

We now present the *Coded Atomic Storage* (CAS) algorithm, which takes advantage of erasure coding techniques to reduce the communication cost for emulating atomic shared memory. CAS is parameterized by an integer k, $1 \leq k \leq N - 2f$; we denote the algorithm with parameter value k by CAS(k). CAS, like ABD and LDR, is a quorum-based algorithm. Later, in Sec. 6, we will present a variant of CAS that has efficient storage costs as well (in addition to having the same communication costs as CAS).

In the ABD algorithm, each message sent by a writer contains a $(tag, value)$ pair. A reader, at the end of its query phase, learns both the tag and the value. This enables easy handling of writer failures since a reader can complete a (failed) write operation[ix]. In contrast, handling of writer failures is not as simple when erasure coding is used.

[ix] In [7], the authors show that readers *must* write, at least in some executions, to ensure atomicity.

write$(value)$

query: Send query messages to all servers asking for the highest tag with label 'fin'; await responses from a quorum.

pre-write: Select the largest tag from the *query* phase; let its integer component be z. Form a new tag t as $(z+1, \text{'id'})$, where 'id' is the identifier of the client performing the operation. Apply the (N, k) MDS code Φ (see Sec. 4) to the value to obtain coded elements $w_1, w_2, \ldots, w_N$. Send $(t, w_s, \text{'pre'})$ to server s for every $s \in \mathcal{N}$. Await responses from a quorum.

finalize: Send a *finalize* message $(t, \text{'null'}, \text{'fin'})$ to all servers. Terminate after receiving responses from a quorum.

read

query: As in the writer protocol.

finalize: Send a *finalize* message with tag t to all the servers requesting the associated coded elements. Await responses from a quorum. If at least k servers include their locally stored coded elements in their responses, then obtain the $value$ from these coded elements by inverting Φ (see Definition 4.1) and terminate by returning $value$.

server

state variable: A variable that is a subset of $\mathcal{T} \times (\mathcal{W} \cup \{\text{'null'}\}) \times \{\text{'pre'}, \text{'fin'}\}$

initial state: Store $(t_0, w_{0,s}, \text{'fin'})$ where s denotes the server and $w_{0,s}$ is the coded element corresponding to server s obtained by apply Φ to the initial value v_0.

On receipt of *query* message: Respond with the highest locally known tag that has a label 'fin', i.e., the highest tag such that the triple $(tag, *, \text{'fin'})$ is at the server, where $*$ can be a coded element or 'null'.

On receipt of *pre-write* message: If there is no record of the tag of the message in the list of triples stored at the server, then add the triple in the message to the list of stored triples; otherwise ignore. Send an acknowledgment.

On receipt of *finalize* from a writer: Let t be the tag associated with the message. If a triple of the form $(t, w_s, \text{'pre'})$ exists in the list of stored triples, then update it to $(t, w_s, \text{'fin'})$. Otherwise add $(t, \text{'null'}, \text{'fin'})$ to the list of stored triples [xvi]. Send an acknowledgment.

On receipt of *finalize* from a reader: Let t be the tag associated with the message. If a triple of the form $(t, w_s, *)$ exists in the list of stored triples where $*$ can be 'pre' or 'fin', then update it to $(t, w_s, \text{'fin'})$ and send (t, w_s) to the reader. Otherwise add $(t, \text{'null'}, \text{'fin'})$ to the list of triples at the server and send an acknowledgment.

Fig. 1. Write, read, and server protocols of the CAS algorithm.

This is because, with an (N, k) MDS code, each message from a writer contains a coded element of the value and therefore, there is no *single* server that has a complete replica of the version. To solve this problem we do not allow readers to observe write operations in progress until enough information has been stored at servers, so that if the writer fails before completing its operation, then a reader can complete that write.

Quorum specification. We define our quorum system, $\mathcal{Q}$, to be the set of all subsets of $\mathcal{N}$ that have at least $\lceil \frac{N+k}{2} \rceil$ elements (server nodes). We refer to the members of $\mathcal{Q}$, as quorum sets. We can show that $\mathcal{Q}$ satisfies the following:

Lemma 5.1. *Suppose that* $1 \leq k \leq N - 2f$. **(i)** *If* $Q_1, Q_2 \in \mathcal{Q}$, *then* $|Q_1 \cap Q_2| \geq k$. **(ii)** *If the number of failed servers is at most* f, *then* $\mathcal{Q}$ *contains at least one quorum set* Q *of non-failed servers.*

The lemma is proved in Appendix C. The CAS algorithm can, in fact, use any quorum system that satisfies properties (i) and (ii) of Lemma 5.1.

Algorithm description. In CAS, we assume that tags are tuples of the form $(z, \text{'id'})$, where z is an integer and 'id' is an identifier of a client node. The ordering on the set of tags $\mathcal{T}$ is defined lexicographically, using the usual ordering on the integers and a predefined ordering on the client identifiers. Fig. 1 contains a description of the read and write protocols, and the server actions of CAS. Here, we provide an overview of the algorithm and, in particular, explain how we handle writer failures.

Each server node maintains a set of $(tag, coded\text{-}element, label)$[x] triples, where we specialize the metadata to $label \in \{\text{'pre'}, \text{'fin'}\}$. The different phases of the write and read protocols are executed sequentially. In each phase, a client sends messages to servers to which the non-failed servers respond. Termination of each phase depends on getting a response from at least one quorum.

The *query* phase is identical in both protocols and it allows clients to discover a recent *finalized object version*, i.e., a recent version with a 'fin' tag. The goal of the *pre-write* phase of a write is to propagate the writer's value to the servers – each server gets a coded element with label 'pre'. Tags associated with pre-written coded elements are not visible to the readers, since the servers respond to *query* messages only with finalized tags. Once a quorum, say Q_{pw}, has acknowledged receipt of the pre-written coded elements, the writer proceeds to its *finalize* phase. In this phase, it propagates a finalize ('fin') label with the tag and waits for a response from a quorum of servers, say Q_{fw}. The purpose of propagating the 'fin' label is to record that the coded elements associated with the tag have been propagated to a quorum[xi]. In fact, when a tag appears anywhere in the system associated with a 'fin' label, it means that the corresponding coded elements reached a quorum Q_{pw} with a 'pre' label at some previous point. The operation of a writer in the two phases following its *query phase* in fact helps overcome the challenge of handling writer failures. In particular, only a tag with the 'fin' label is visible to the readers. This means that any tag obtained by a reader in the query phase has been finalized, which indicates that the corresponding coded elements have been propagated to at least one quorum. The reader is guaranteed to get k unique coded units from any quorum of non-failed nodes, because such a quorum has an intersection of k nodes with Q_{pw}[xii]. Finally, the reader helps propagate the tag to a quorum, and this helps complete (possibly failed) writes as well. We next state the man result of this section.

Theorem 5.2. *CAS emulates shared atomic read/write memory.*

To prove Theorem 5.2, we need to show atomicity, Lemma 5.3, and liveness, Lemma 5.9.

Lemma 5.3 (Atomicity). *CAS(k) is atomic.*

To show Lemma 5.3 we show that it satisfies properties of the following lemma, which suffices to establish atomicity.

[x] The 'null' entry indicates that no coded element is stored, only a label is stored with the tag; the storage cost associated storing a null coded element is assumed to be negligible.

[xi] It is worth noting that Q_{fw} and Q_{pw} need not be the same quorum. That is, for a tag, one quorum of servers, Q_{pw} may have the actual coded elements, some of them with the 'pre' label and others with the 'fin' label; a different quorum Q_{fw} may have the 'fin' label, some of them with the associated coded element and others with the 'null' entry for the coded element.

[xii] We note that any server s in $Q_{pw} \cap Q_{fw}$ responds to the read's *finalize* message with the locally stored coded element w_s.

Lemma 5.4. *(Paraphrased Lemma 13.16 [15].) Suppose that the environment is well-behaved, meaning that an operation is invoked at a client only if no other operation was performed by the client, or the client received a response to the last operation it initiated. Let β be a (finite or infinite) execution of a read/write object, where β consists of invocations and responses of read and write operations and where all operations terminate. Let Π be the set of all operations in β.*

Suppose that $\prec$ is an irreflexive partial ordering of all the operations in Π, satisfying the following properties: **(1)** *If the response for π_1 precedes the invocation for π_2 in β, then it cannot be the case that $\pi_2 \prec \pi_1$.* **(2)** *If π_1 is a write operation in Π and π_2 is any operation in Π, then either $\pi_1 \prec \pi_2$ or $\pi_2 \prec \pi_1$.* **(3)** *The value returned by each read operation is the value written by the last preceding write operation according to $\prec$ (or v_0, if there is no such write).*

The following definition will be useful for proving Lemma 5.4.

Definition 5.5. *Consider an execution β of CAS and consider an operation π that terminates in β. The* tag *of operation π, denoted as $T(\pi)$, is defined as follows: If π is a read, then, $T(\pi)$ is the highest tag received in its* query *phase. If π is a write, then, $T(\pi)$ is the new tag formed in its* pre-write *phase.*

We show Lemma 5.4 by defining the partial order $\prec$ on operations based on their tags. To do so, in Lemmas 5.6, 5.7, and 5.8, we show certain properties satisfied by the tags. Specifically, we show in Lemma 5.6 that, in any execution of CAS, at any point after an operation π terminates, the tag $T(\pi)$ has been propagated with the 'fin' label to at least one quorum of servers. Intuitively speaking, Lemma 5.6 means that if an operation π terminates, the tag $T(\pi)$ is visible to any operation that is invoked after π terminates. We crystallize this intuition in Lemma 5.7, where we show that any operation that is invoked after an operation π terminates acquires a tag that is at least as large as $T(\pi)$. Using Lemma 5.7 we show Lemma 5.8, which states that the tag acquired by each write operation is unique. Finally, we show that Lemmas 5.7 and 5.8 imply Lemma 5.4 by defining that partial order $\prec$ on operations based on the tags acquired by the operations. Here, we present only a proof of Lemma 5.6, and a brief sketch of a proof of Lemma 5.4. We present formal proofs of Lemmas 5.7, 5.8, and 5.3 in Appendix D.

Lemma 5.6. *In any execution β of CAS, for an operation π that terminates in β, there exists a quorum $Q_{fw}(\pi)$ such that the following is true at every point of the execution β after π terminates: Every server of $Q_{fw}(\pi)$ has $(t, *, \text{'}\mathrm{fin}\text{'})$ in its set of stored triples, where $*$ is either a coded element or 'null', and $t = T(\pi)$.*

Proof. The proof is the same whether π is a read or a write operation. The operation π terminates after completing its *finalize* phase, during which it receives responses from a quorum, say $Q_{fw}(\pi)$, to its *finalize* message. This means that every server s in $Q_{fw}(\pi)$ responded to the *finalize* message from π at some point before the point of termination of π. From the server protocol, we can observe that every server s in $Q_{fw}(\pi)$ stores the triple $(t, *, \text{'}\mathrm{fin}\text{'})$ at the point of responding to the *finalize* message of π, where $*$ is either a coded element or 'null'. Furthermore, the server s stores the triple at every point after the point of responding to the *finalize* message of π and hence at every point after the point of termination of π. □

Lemma 5.7. *Consider any execution β of CAS, and let π_1, π_2 be two operations that terminate in β. Suppose that π_1 returns before π_2 is invoked. Then $T(\pi_2) \geq T(\pi_1)$. Furthermore, if π_2 is a write operation, then $T(\pi_2) > T(\pi_1)$.*

Lemma 5.8. *Let π_1, π_2 be write operations that terminate in an execution β of CAS. Then $T(\pi_1) \neq T(\pi_2)$.*

Intuitively, Lemma 5.7 follows from Lemma 5.6 since, tag $T(\pi_1)$ has been propagated to a quorum with the 'fin' label implying that operation π_2 receives a tag that is at least as large as $T(\pi_1)$. We next provide a brief sketch of the proof of Lemma 5.3. *Proof Sketch of Lemma 5.3.* In any execution β of CAS, we order operations π_1, π_2 as $\pi_1 \prec \pi_2$ if **(i)** $T(\pi_1) < T(\pi_2)$, or **(ii)** $T(\pi_1) = T(\pi_2)$, π_1 is a write and π_2 is a read. It can be verified that $\prec$ is a partial order on the operations of β. With this ordering, the first two properties of Lemma 5.4 follow from Lemmas 5.7 and 5.8 respectively. The last property of Lemma 5.4 follows from examination of the CAS algorithm, in particular, on noting that a read that terminates always returns the value associated with its tag. □

Lemma 5.9 (Liveness). *CAS(k) satisfies the following* liveness *condition: If $1 \leq k \leq N - 2f$, then every operation terminates in every fair execution of CAS(k) where the number of failed server nodes is no bigger than f .*

We provide only a part of the proof here; the remaining parts appear in Appendix E.
Proof (Partial.): By examination of the algorithm we observe that termination of any operation depends on termination of its phases. So, to show liveness, we need to show that each phase of each operation terminates. Termination of a write operation and the *query* phase of a read are contingent on receiving responses from a quorum of non-failed servers in the execution; property (**ii**) of Lemma 5.1 guarantees the existence of such a quorum, and thus ensures their termination (see Appendix E for more details).

We show the termination of a reader's *finalize* phase here since it is more challenging. By using property (**ii**) of Lemma 5.1, we can show that a quorum, say Q_{fw} of servers responds to a reader's *finalize* message. For the *finalize* phase of a read to terminate, there is an additional requirement that at least k servers include coded elements in their responses. To show that this requirement is satisfied, suppose that the read acquired a tag t in its *query* phase. From examination of the CAS algorithm, we can infer that, at some point before the point of termination of the read's *query* phase, a writer propagated a *finalize* message with tag t. Let us denote by $Q_{pw}(t)$, the set of servers that responded to this write's *pre-write* phase. Now, we argue that all servers in $Q_{pw}(t) \cap Q_{fw}$ respond to the reader's *finalize* message with a coded element. To see this, let s be any server in $Q_{pw}(t) \cap Q_{fw}$. Since s is in $Q_{pw}(t)$, the server protocol for responding to a *pre-write* message implies that s has a coded element, w_s, at the point where it responds to that message. Since s is in Q_{fw}, it also responds to the reader's *finalize* message, and this happens at some point after it responds to the *pre-write* message. So it responds with its coded element w_s. From Lemma 5.1, it is clear that $|Q_{pw}(t) \cap Q_{fw}| \geq k$ implying that the reader receives at least k coded elements in its *finalize* phase and hence terminates.

□

Cost analysis. We analyze the communication cost of CAS next.

Theorem 5.10. *The write and read communication costs of the CAS(k) are both equal to $N/k \log_2 |\mathcal{V}|$ bits.*

Proof. For either protocol, to measure the communication cost, observe that messages carry coded elements which have size $\frac{\log_2 |\mathcal{V}|}{k}$ bits. More formally, each message is an element from $\mathcal{T} \times \mathcal{W} \times \{\text{'pre'}, \text{'fin'}\}$, where, $\mathcal{W}$ is a coded element corresponding to one of the N outputs of the MDS code Φ. As described in Sec. 4, $\log_2 |\mathcal{W}| = \frac{\log_2 |\mathcal{V}|}{k}$. The only messages that incur a non-negligible communication cost are the messages sent from the client to the servers in the *pre-write* phase of a write and the messages sent from the servers to a client in the *finalize* phase of a read. It can be seen that the total communication cost of read and write operations of the CAS algorithm are $\frac{N}{k} \log_2 |\mathcal{V}|$ bits, that is, they are upper bounded by this quantity and the said costs are incurred in certain worst-case executions. □

Since k is a parameter that can be freely chosen so that $1 \leq k \leq N - 2f$, the communication cost for both reads and writes be made as small as $\frac{N}{N-2f} \log_2 |\mathcal{V}|$ bits by choosing $k = N - 2f$ (see Tab. 1).

6 Storage Optimized Variant of CAS

In this section, we present a variant of the CAS algorithm that has bounded storage cost provided that the number of operations that are concurrent to any read operation is bounded. The key idea that enables bounded storage cost is *garbage collection* of sufficiently old coded elements, which do not impact the external behavior of read and write operations[xiii]. We begin by describing our variant, the CAS with Garbage Collection (CASGC) algorithm. The CASGC algorithm assumes that the number of operations that are concurrent with a read operation is upper bounded by a parameter δ, whose value is known to all the servers in the system. Like CAS, CASGC is parametrized by an integer k, where $1 \leq k \leq N - 2f$; we will denote the algorithm with the parameter value k as CASGC(k). After describing the algorithm, we will show that CASGC satisfies the desired correctness requirements (atomicity and liveness), in addition to having a storage cost of $(\delta + 1)\frac{N}{k} \log_2 |\mathcal{V}|$.

The CASGC algorithm is essentially the same as CAS with an additional garbage collection step at the servers. In particular, the only differences between the two algorithms lie in the set of server actions in response to receiving a *pre-write* message and a *finalize* message from a reader. The server actions in the CASGC algorithm are described in Fig. 2. The figure shows that, in CASGC, each server stores the latest $\delta + 1$ coded elements along with their tags and their labels; it stores only metadata related to the other tags it has seen. On receiving a *pre-write* message, it performs a garbage collection step before responding to the message. The garbage collection step checks if the server has more than $\delta + 1$ coded elements; if so, it replaces the triple $(t', coded\text{-}element, label)$ by $(t', \text{'null'}, label)$ where t' is the smallest tag associated with a coded element in the list of triples stored at the server. This strategy ensures that the server stores at most $\delta + 1$ coded elements at any point. If a reader requests, through a *finalize* message, a coded element that is already garbage collected, the server simply ignores this request.

Now we bound the storage cost of the CASGC algorithm, and show that CASGC satisfies atomicity and liveness.

[xiii] Recall that the LDR algorithm, much like the CAS algorithm has unbounded storage cost. The garbage collection technique presented here could perhaps be utilized on the LDR algorithm to bound its storage cost as well.

servers

state variable: A variable that is a subset of $\mathcal{T} \times (\mathcal{W} \cup \{\text{'null'}\}) \times \{\text{'pre'}, \text{'fin'}, (\text{'pre'}, \text{'gc'}), (\text{'fin'}, \text{'gc'})\}$
initial state: Same as in Fig. 1.

On receipt of *query* message: Similar to Fig. 1, respond with the highest locally available tag labeled 'fin', i.e., Respond with the highest tag such that the triple $(tag, *, \text{'fin'})$ or $(tag, \text{'null'}, (\text{'fin'}, \text{'gc'}))$ is at the server, where $*$ can be a coded element or 'null'.

On receipt of a *pre-write* message: Perform the actions as described in Fig. 1 except the sending of an acknowledgment. Then, perform *garbage collection*. Then, send an acknowledgment.

On receipt of *finalize* from a writer: Let t be the tag associated with the message. If a triple of the form $(t, \text{'null'}, (\text{'fin'}, \text{'gc'}))$ exists in the list of stored triples, send an acknowledgement. Otherwise perform steps as in Fig. 1.

On receipt of a *finalize* message from a reader: Let t be the tag associated with the message. If $(t, \text{'null'}, (*, \text{'gc'}))$ exists in the list of locally available triples where $*$ can be either 'fin' or 'pre', then ignore the incoming message. Otherwise, perform steps as in Fig. 1.

garbage collection: If there are more than $\delta + 1$ triples with coded elements, then, for every tag t' such that the server has the triple $(t', coded\text{-}element, label)$, and t' is not one of the highest $\delta + 1$ tags at the server with a coded element, replace the triple by $(t', \text{'null'}, (label, \text{'gc'}))$.

Fig. 2. Server Actions for the CASGC algorithm.

Theorem 6.1. *The write and read communication costs of CASGC(k) are each equal to* $\frac{N}{k} \log_2 |\mathcal{V}|$ *bits. The storage cost of CASGC(k) is* $(\delta + 1)\frac{N}{k} \log_2 |\mathcal{V}|$ *bits.*

Proof. The proofs of the write and read communication costs of the CASGC algorithm are similar to the proofs the costs for the CAS algorithm. We omit the proof here for brevity. We can show that the storage cost of the CASGC algorithm is $(\delta + 1)\frac{N}{k} \log_2 |\mathcal{V}|$ bits by noting that each server stores at most $(\delta + 1)$ coded elements, and by noting that in certain executions, all N servers simultaneously store $(\delta + 1)$ coded elements. □

Since k can be chosen to be any parameter in the rage $1 \leq k \leq N - 2f$, the storage cost of the CASGC algorithm can be made as small as $\frac{(\delta+1)N}{N-2f} \log_2 |\mathcal{V}|$ bits (see Tab. 1).

Theorem 6.2. *If the number of operations concurrent to any read operation is upper bounded by* δ*, then the CASGC algorithm satisfies atomicity and liveness.*

To show the theorem, we observe that, from the perspective of the clients, the only difference between CAS and CASGC is in the server response to a read's *finalize* message. In CASGC, when a coded element has been garbage collected, a server ignores a read's *finalize* message. If the number of write operations concurrent with a read are bounded by δ, we show the following key property of CASGC: at any point when a server ignores a reader's *finalize* message because of garbage collection, the read operation that sent the message has already terminated. We show this formally in Lemma 6.3. The lemma implies that garbage collection does not influence the external behavior of the operations, and therefore, the proof of correctness of CASGC follows along the same lines as CAS. We state and prove Lemma 6.3 here. We present a more formal proof of Theorem 6.2 via a simulation relation to CAS in Appendix F.

Lemma 6.3. *Consider any execution* α *of CASGC where the number of operations concurrent to any read operation is upper bounded by* δ*. Consider any point of* α *where*

the triple $(t, \text{'null'}, (*, \text{'gc'}))$ *is in the set of triples stored at some server, where* $*$ *could be* 'pre' *or* 'fin'. *If a read operation in* α *acquired tag* t *in its* query *phase, then the operation terminated before this point.*

Proof. We show the lemma by contradiction. Consider an execution α of CASGC and consider a point of α (if such a point exists) where: **(i)** a server s has $(t, \text{'null'}, (*, \text{'gc'}))$ in its set of triples, where $*$ is 'pre' or 'fin', and **(i)** a read operation that selected tag t in its *query* phase has not terminated at this point. Because server s has garbage collected tag t, the server protocol of CASGC implies that server s received at least $\delta + 1$ coded elements corresponding to tags that are higher than t before the point in consideration. We denote the tags by $t_1, t_2, \ldots, t_{\delta+1}$, where, $t_i > t, i = 1, 2, \ldots, \delta + 1$. For any i in $\{1, 2, \ldots, \delta + 1\}$, the presence of a tag t_i at server s means that a write operation, say π_i, must have committed to tag t_i in its *pre-write* phase before this point in α. Since, at this point, the read with tag t has not terminated, there are only two possibilities.

(a) All the write operations $\pi_1, \pi_2, \ldots, \pi_{\delta+1}$ are concurrent with the reader.
(b) At least one of the operations π_i terminated before the read operation started.

To show the lemma, it suffices to show that neither (a) nor (b) is possible. (a) violates the concurrency bound of δ and is therefore impossible. To show that (b) cannot occur, we use an argument similar to the proof of the first property of Lemma 5.4 in Sec. 5. In particular, we can show[xiv] that the following property, analogous to Lemma 5.6, holds for CASGC: After operation π_i terminates, a quorum $Q_{fw}(\pi_i)$ exists where each server of the quorum has an element $(t_i, *, \text{'fin'})$ or $(t_i, \text{'null'}, (\text{'fin'}, \text{'gc'}))$ where $*$ can be 'null' or a coded element. This implies that the reader must have acquired a tag $t \geq t_i$ during its query phase contradicting our earlier assumption. Hence, (b) is impossible.

□

7 Conclusions

In this paper, we have proposed a new approach toward emulation of atomic shared memory in asynchronous message-passing systems. Contributions of our paper apply to two different fields. First, we analyze communication and storage costs of traditional techniques in distributed computing theory; we also provide new coding-based algorithms that outperform traditional techniques with respect to these costs. Second, for the area of information and coding theory, which traditionally studies costs of communication and storage techniques under certain classical constraints (tolerance against *erasures* (failures)), we introduce the new constraint of ensuring consistency. In particular, we study techniques that ensure consistency in addition to failure tolerance, and their associated costs.

Broadly, understanding the performance of communication and storage costs of consistent distributed shared-memory systems is an open and ripe research area. In this paper, we considered a point-to-point reliable message passing system where the cost of a message is equal to the size of the values it contains. In our model, an open question is whether the shared memory algorithms of this paper are optimal; such a study will need the construction of lower bounds on the costs incurred. More generally, it is of relevance, both to theory and practice, to explore the performance of other models for message-passing architectures such as wireless systems, packet erasure channels, and wireline communication networks.

[xiv] The proof is almost identical to the proof of Lemma 5.6 and is omitted here for brevity.

References

1. Common RAID disk data format specification, March 2009.
2. A. Agrawal and P. Jalote. Coding-based replication schemes for distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 6(3):240 –251, March 1995.
3. M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58:7:1–7:32, April 2011.
4. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, pages 363–375, New York, NY, USA, 1990. ACM.
5. M. Blaum, P. Farrell, and H. Van Tilborg. Array codes, handbook of coding theory. *Elsevier Science*, 2:1855–1909, 1998. Chapter 22, Editors: V.S. Pless and W.C. Huffman.
6. Y. Cassuto, A. Datta, and F. Oggier. Coding for distributed storage, March 2013. SIGACT News.
7. R. Fan and N. Lynch. Efficient replication of large data objects. In *In Proceedings of the 17th International Symposium on Distributed Computing (DISC)*, pages 75–91, 2003.
8. A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Trans. Comput. Syst.*, 19(2):171–216, 2001.
9. D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM.
10. S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, December 2010.
11. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.
12. L. Lamport. On interprocess communication. Part I: Basic formalism. *Distributed Computing*, 2(1):77–85, 1986.
13. S. Lin and D. J. Costello. *Error Control Coding, Second Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.
14. N. Lynch and A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *In Symposium on Fault-Tolerant Computing*, pages 272–281. IEEE, 1997.
15. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.
16. D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, October 1998.
17. J. S. Plank. T1: erasure codes for storage applications. In *Proc of the 4th USENIX Conference on File and Storage Technologies. San Francisco*, pages 1–74, 2005.
18. I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.
19. R. Roth. *Introduction to coding theory*. Cambridge University Press, 2006.
20. R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.
21. M. Vukolić. Quorum systems: With applications to storage and consensus. *Synthesis Lectures on Distributed Computing Theory*, 3(1):1–146, 2012/03/01 2012.

A Descriptions of the ABD and LDR Algorithms

Here, we describe the ABD and LDR algorithms, and evaluate their communication and storage costs. We present the ABD and LDR algorithms in Fig. 3 and Fig. 4 respectively.

write$(value)$

get: Send query request to all servers, await (tag) responses from a majority of server nodes. Select the largest tag; let its integer component be z. Form a new tag t as $(z+1,$ 'id'$)$, where 'id' is the identifier of the client performing the operation.

put: Send the pair $(t, value)$ to all servers, await acknowledgment from a majority of server nodes, and then terminate.

read

get: Send query request to all servers, await $(tag, value)$ responses from a majority. Select a tuple with the largest tag, say (t, v).

put: Send (t, v) to all servers, await acknowledgment from a majority, and then terminate by returning the value v.

server

state variable: A variable which contains an element of $\mathcal{T} \times \mathcal{V}$
initial state: Store the default $(tag, value)$ pair (t_0, v_0).

On receipt of *get* message from a read: Respond with the locally available $(tag, value)$ pair.

On receipt of *get* message from a write: Respond with the locally available tag.

On receipt of *put* message: If the tag of the message is higher than the locally available tag, store the $(tag, value)$ pair of the message at the server. In any case, send an acknowledgment.

Fig. 3. Write, read, and server protocols of the ABD algorithm.

Communication and Storage costs of ABD and LDR algorithms.
Proof of Theorem 3.1. We first present arguments that upper bound the communication and storage cost for every execution of the ABD algorithm. The ABD algorithm presented here is fitted to our model. Specifically in [4, 14] there is no clear cut separation between clients and servers. However, this separation does not change the costs of the algorithm. Then we present worst-case executions that incur the costs as stated in the theorem.
Upper bounds: First consider the write protocol. It has two phases, *get* and *put*. The *get* phase of a write involves transfer of a tag, but not of actual data, and therefore has negligible communication cost. In the *put* phase of a write, the client sends a value from the set $\mathcal{T} \times \mathcal{V}$ to every server node; the total communication cost of this phase is at most $N \log_2 |\mathcal{V}|$ bits. Therefore the total write communication cost is at most $N \log_2 |\mathcal{V}|$ bits. In the *get* phase of the read protocol, the message from the client to the servers contains only metadata, and therefore has negligible communication cost. However, in this phase, each of the N servers could respond to the client with a message from $\mathcal{T} \times \mathcal{V}$; therefore the total communication cost of the messages involved in the *get* phase is upper bounded

by $N \log_2 |\mathcal{V}|$ bits. In the *put* phase of the read protocol, the read sends an element of $\mathcal{T} \times \mathcal{V}$ to N servers. Therefore, this phase incurs a communication cost of at most $N \log_2 |\mathcal{V}|$ bits. The total communication cost of a read is therefore upper bounded by $2N \log_2 |\mathcal{V}|$ bits.

The storage cost of ABD is no bigger than $N \log_2 |\mathcal{V}|$ bits because each server stores at most one value - the latest value it receives.

Worst-case executions: Informally speaking, due to asynchrony and the possibility of failures, clients always send requests to all servers and in the worst case, all servers respond. Therefore the upper bounds described above are tight.

For the write protocol, the client sends the value to all N nodes in its *put* phase. So the write communication cost in an execution where at least one write terminates is $N \log_2 |\mathcal{V}|$ bits. For the read protocol, consider the following execution, where there is one read operation, and one write operation that is concurrent with this read. We will assume that none of the N servers fail in this execution. Suppose that the writer completes its get phase, and commits to a tag t. Note that t is the highest tag in the system at this point. Suppose that among the N messages that the writer sends in its put phase with the value and tag t, Now the writer begins its put phase where it sends N messages with the value and tag t. At least one of these messages, say the message to server 1, arrives.the remaining messages are delayed, i.e., they are assumed to reach after the portion of the execution segment described here. At this point, the read operation begins and receives $(tag, value)$ pairs from all the N server nodes in its get phase. Of these N messages, at least one message contains the tag t and the corresponding value. Note that t is the highest tag it receives. Therefore, the put phase of the read has to sends N messages with the tag t and the corresponding value - one message to each of the N servers that which responded to the read in the get phase with an older tag.

The read protocol has two phases. The cost of a read operation in an execution is the sum of the communication costs of the messages sent in its *get* phase and those sent in its *put* phase. The *get* phase involves communication of N messages from $\mathcal{T} \times \mathcal{V}$, one message from each server to the client, and therefore incurs a communication cost of $N \log_2 |\mathcal{V}|$ bits provided that every server is active. The *put* phase involves the communication of a message in $\mathcal{T} \times \mathcal{V}$ from the client to every server thereby incurring a communication cost of $N \log_2 |\mathcal{V}|$ bits as well. Therefore, in any execution where all N servers are active, the communication cost of a read operation is $2N \log_2 |\mathcal{V}|$ bits and therefore the upper bound is tight.

The storage cost is equal to $N \log_2 |\mathcal{V}|$ bits since each of the N servers store exactly one value from $\mathcal{V}$. □

Proof of Theorem 3.2.

Upper bounds: In LDR servers are divided into two groups: *directory* servers used to manage object metadata, and *replication* servers used for object replication. Read and write protocols have three sequentially executed phases. The *get-metadata* and *put-metadata* phases incur negligible communication cost since only metadata is sent over the message-passing system. In the *put* phase, the writer sends its messages, each of which is an element from $\mathcal{T} \times \mathcal{V}$, to $2f + 1$ replica servers and awaits $f + 1$ responses; since the responses have negligible communication cost, this phase incurs a total communication cost of at most $(2f + 1) \log_2 |\mathcal{V}|$ bits. The read protocol is less taxing, where the reader during the *get* phase queries $f + 1$ replica servers and in the worst case, all

write$(value)$

get-metadata: Send query request to directory servers, and await $(tag, location)$ responses from a majority of directory servers. Select the largest tag; let its integer component be z. Form a new tag t as $(z+1,$ 'id'$)$, where 'id' represents the identifier of the client performing the operation.

put: Send $(t, value)$ to $2f + 1$ replica servers, await acknowledgment from $f + 1$. Record identifiers of the first $f + 1$ replica servers that respond, call this set of identifiers $\mathcal{S}$.

put-metadata: Send $(t, \mathcal{S})$ to all directory servers, await acknowledgment from a majority, and then terminate.

read

get-metadata: Send query request to directory servers, and await $(tag, location)$ responses from a majority of directory servers. Choose a $(tag, location)$ pair with the largest tag, let this pair be $(t, \mathcal{S})$.

put-metadata: Send $(t, \mathcal{S})$ to all directory servers, await acknowledgment from a majority.

get: Send *get object* request to any $f + 1$ replica servers recorded in $\mathcal{S}$ for tag t. Await a single response and terminate by returning a value.

replica server

state variable: A variable that is subset of $\mathcal{T} \times \mathcal{V}$

initial state: Store the default $(tag, value)$ pair (t_0, v_0).

On receipt of *put* message: Add the $(tag, value)$ pair in the message to the set of locally available pairs. Send an acknowledgment.

On receipt of *get* message: If the value associated with the requested tag is in the set of pairs stored locally, respond with the value. Otherwise ignore.

directory server

state variable: A variable that is an element of $\mathcal{T} \times 2^{\mathcal{R}}$ where $2^{\mathcal{R}}$ is the set of all subsets of $\mathcal{R}$.

initial state: Store $(t_0, \mathcal{R})$, where $\mathcal{R}$ is the set of all replica servers.

On receipt of *get-metadata* message: Send the $(tag, \mathcal{S})$ be the pair stored locally.

On receipt of *put-metadata* message: Let $(t, \mathcal{S})$ be the incoming message. At the point of reception of the message, let $(tag, \mathcal{S}_1)$ be the pair stored locally at the server. If t is equal to the tag stored locally, then store $(t, \mathcal{S} \cup \mathcal{S}_1)$ locally. If t is bigger than tag and if $|\mathcal{S}| \geq f+1$, then store $(t, \mathcal{S})$ locally. Send an acknowledgment.

Fig. 4. Write, read, and server protocols of the LDR algorithm

respond with a message containing an element from $\mathcal{T} \times \mathcal{V}$ thereby incurring a total communication cost of at most $(f + 1) \log_2 |\mathcal{V}|$ bits.

Worst-case executions: It is clear that in every execution where at least one writer terminates, the writer sends out $(2f + 1)$ messages to replica servers that contain the value, thus incurring a write communication cost of $(2f+1) \log_2 |\mathcal{V}|$ bits. Similarly, for a read, in certain executions, all $(f + 1)$ replica servers that are selected in the *put phase* of the read respond to the *get* request from the client. So the upper bounds derived above are tight. □

B Discussion on Erasure Codes

As described in Sec. 4, much literature in coding theory involves the design of (N, k) codes for which the redundancy factor[xv] can be made as small as possible. In the classical erasure coding model, the extent to which the redundancy factor can be reduced depends on f - the maximum number of server failures that are to be tolerated. In particular, an (N, k) MDS code, when employed to store the value of the data object, tolerates $N - k$ server node failures; this is because the definition of an MDS code implies that the data can be recovered from any k surviving nodes. Thus, for an N-server system that uses an MDS code, we must have $k \leq N - f$, meaning that the redundancy factor is at least $\frac{N}{N-f}$. It is well known [19] that, given N and f, the parameter k cannot be made larger than $N - f$ so that the redundancy factor is lower bounded by $\frac{N}{N-f}$ for *any* code even if it is not an MDS code; In fact, an MDS code can equivalently be defined as one which attains this lower bound on the redundancy factor. In coding theory, this lower bound is known as the Singleton bound [19]. Given parameters N, k, the question of whether an (N, k) MDS code exists depends on the alphabet of code $\mathcal{W}$. We next discuss some of the relevant assumptions that we (implicitly) make in this paper to enable the use of an (N, k) MDS code in our algorithms.

B.1 Assumption on $|\mathcal{V}|$ due to Erasure Coding

Recall that, in our model, each value v of a data object belongs to a finite set $\mathcal{V}$. In our system, for the use of coding, we assume that $\mathcal{V} = \mathcal{W}^k$ for some finite set $\mathcal{W}$ and that $\Phi : \mathcal{W}^k \rightarrow \mathcal{W}^N$ is an MDS code. Here we refine these assumptions using classical results from erasure coding theory. In particular, the following result is useful.

Theorem B.2. *Consider a finite set $\mathcal{W}$ such that $|\mathcal{W}| \geq N$. Then, for any integer $k < N$, there exists an (N, k) MDS code $\Phi : \mathcal{W}^k \rightarrow \mathcal{W}^N$.*

One proof for the above in coding theory literature is constructive. Specifically, it is well known that when $|\mathcal{W}| \geq N$, then Φ can be constructed using the Reed-Solomon code construction [18, 19, 13]. The above theorem implies that, to employ a Reed-Solomon code over our system, we shall need the following two assumptions:

- k divides $\log_2 |\mathcal{V}|$, and
- $\log_2 |\mathcal{V}|/k \geq \log_2 N$.

Thus all our results are applicable under the above assumptions.

In fact, the first assumption above can be replaced by a different assumption with only a negligible effect on the communication and storage costs. Specifically, if $\log_2 |\mathcal{V}|$ were not a multiple of k then, one could pad the value with $\left(\lceil \frac{\log_2 |\mathcal{V}|}{k} \rceil k - \log_2 |\mathcal{V}|\right)$ "dummy" bits, all set to 0, to ensure that the (padded) object has a size that is multiple of k; note that this padding is an overhead. The size of the padded object would be

[xv] Literature in coding theory literature often studies the *rate* $\frac{N}{k}$ of a code, which is the reciprocal of the redundancy factor, i.e., the rate of an (N, k) code is $\frac{k}{N}$. In this paper, we use the redundancy factor in our discussions since it enables a somewhat more intuitive connection with the costs of our algorithms in Tab. 1 and Theorems 3.1, 3.2, 5.10, 6.1.

$\lceil \frac{\log_2 |\mathcal{V}|}{k} \rceil k$ bits and the size of each coded element would be $\lceil \frac{\log_2 |\mathcal{V}|}{k} \rceil$ bits. If we assume that $\log_2 |\mathcal{V}| \gg k$ then, $\lceil \frac{\log_2 |\mathcal{V}|}{k} \rceil \approx \frac{\log_2 |\mathcal{V}|}{k}$ meaning that the padding overhead can be neglected. Consequently, the first assumption can be replaced by the assumption that $\log_2 |\mathcal{V}| \gg k$ with only a negligible effect on the communication and storage costs.

C Proof of Lemma 5.1

Proof of property (**i**): By the definition, each $Q \in \mathcal{Q}$ has cardinality at least $\lceil \frac{N+k}{2} \rceil$. Therefore, for $Q_1, Q_2 \in \mathcal{Q}$, we have

$$\begin{aligned} |Q_1 \cap Q_2| &= |Q_1| + |Q_2| - |Q_1 \cup Q_2| \\ &\geq 2 \left\lceil \frac{N+k}{2} \right\rceil - |Q_1 \cup Q_2| \\ &\overset{(a)}{\geq} 2 \left\lceil \frac{N+k}{2} \right\rceil - N \geq k, \end{aligned}$$

where we have used the fact that $|Q_1 \cup Q_2| \leq N$ in (a).
Proof of property (**ii**): Let $\mathcal{B}$ be the set of all the server nodes that fail in an execution, where $|\mathcal{B}| \leq f$. We need to show that there exists at least one quorum set $Q \in \mathcal{Q}$ such that $Q \subseteq \mathcal{N} - \mathcal{B}$, that is, at least one quorum survives. To show this, because of the definition of our quorum system, it suffices to show that $|\mathcal{N} - \mathcal{B}| \geq \lceil \frac{N+k}{2} \rceil$. We show this as follows:

$$|\mathcal{N} - \mathcal{B}| \geq N - f \overset{(b)}{\geq} N - \left\lfloor \frac{N-k}{2} \right\rfloor = \left\lceil \frac{N+k}{2} \right\rceil,$$

where, (b) follows because $k \leq N - 2f$ implies that $f \leq \lfloor \frac{N-k}{2} \rfloor$.

D Atomicity of CAS: Remaining Proofs

In this section, we first prove Lemmas 5.7 and 5.8, and then Lemma 5.3.
Proof of Lemma 5.7. To establish the lemma, it suffices to show that the tag acquired in the *query* phase of π_2, denoted as $\hat{T}(\pi_2)$, is at least as big as $T(\pi_1)$, that is, it suffices to show that $\hat{T}(\pi_2) \geq T(\pi_1)$. This is because, by examination of the client protocols, we can observe that if π_2 is a read, $T(\pi_2) = \hat{T}(\pi_2)$, and if π_2 is a write, $T(\pi_2) > \hat{T}(\pi_2)$.

To show that $\hat{T}(\pi_2) \geq T(\pi_1)$ we use Lemma 5.6. We denote the quorum of servers that respond to the *query* phase of π_2 as $\hat{Q}(\pi_2)$. We now argue that every server s in $\hat{Q}(\pi_2) \cap Q_{fw}(\pi_1)$ responds to the *query* phase of π_2 with a tag that is at least as large as $T(\pi_1)$. To see this, since s is in $Q_{fw}(\pi_1)$, Lemma 5.6 implies that s has a tag $T(\pi_1)$ with label 'fin' at the point of termination of π_1. Since s is in $\hat{Q}(\pi)$, it also responds the *query* message of π_2, and this happens at some point after the termination of π_1 because π_2 is invoked after π_1 responds. From the server protocol, we can infer that server s responds to the *query* message of π_2 with a tag that is no smaller than $T(\pi_1)$. Because of Lemma 5.1, there is at least one server s in $\hat{Q}(\pi_2) \cap Q_{fw}(\pi_1)$ implying that operation π_2 receives at least one response in its *query* phase with a tag that is no smaller than $T(\pi_1)$. Therefore $\hat{T}(\pi_2) \geq T(\pi_1)$. □

Proof of Lemma 5.8. Let π_1, π_2 be two write operations that terminate in execution β. Let C_1, C_2 respectively indicate the identifiers of the client nodes at which operations π_1, π_2 are invoked. We consider two cases.
Case 1, $C_1 \neq C_2$: From the write protocol, we note that $T(\pi_i) = (z_i, C_i)$. Since $C_1 \neq C_2$, we have $T(\pi_1) \neq T(\pi_2)$.
Case 2, $C_1 = C_2$: Recall that operations at the same client follow a "handshake" discipline, where a new invocation awaits the response of a preceding invocation. This means that one of the two operations π_1, π_2 should complete before the other starts. Suppose that, without loss of generality, the write operation π_1 completes before the write operation π_2 starts. Then, Lemma 5.7 implies that $T(\pi_2) > T(\pi_1)$. This implies that $T(\pi_2) \neq T(\pi_1)$. □

Proof of Lemma 5.3. Recall that we defined our ordering $\prec$ as follows: In any execution β of CAS, we order operations π_1, π_2 as $\pi_1 \prec \pi_2$ if **(i)** $T(\pi_1) < T(\pi_2)$, or **(ii)** $T(\pi_1) = T(\pi_2)$, π_1 is a write and π_2 is a read.

We first verify that the above ordering is a partial order, that is, if $\pi_1 \prec \pi_2$, then it cannot be that $\pi_2 \prec \pi_1$. We prove this by contradiction. Suppose that $\pi_1 \prec \pi_1$ and $\pi_2 \prec \pi_1$. Then, by definition of the ordering, we have that $T(\pi_1) \leq T(\pi_2)$ and vice-versa, implying that $T(\pi_1) = T(\pi_2)$. Since $\pi_1 \prec \pi_2$ and $T(\pi_1) = T(\pi_2)$, we have that π_1 is a write and π_2 is a read. But a symmetric argument implies that π_2 is a write and π_1 is a read, which is a contradiction. Therefore $\prec$ is a partial order.

With the ordering $\prec$ defined as above, we now show that the three properties of Lemma 5.4 are satisfied. For property $(\mathbf{1})$, consider an execution β and two distinct operations π_1, π_2 in β such that π_1 returns before π_2 is invoked. If π_2 is a read, then Lemma 5.7 implies that $T(\pi_2) \geq T(\pi_1)$. By definition of the ordering, it cannot be the case that $\pi_2 \prec \pi_1$. If π_1 is a write, then Lemma 5.7 implies that $T(\pi_2) > T(\pi_1)$ and so, $\pi_1 \prec \pi_2$. Since $\prec$ is a partial order, it cannot be the case that $\pi_2 \prec \pi_1$.

Property $(\mathbf{2})$ follows from the definition of the $\prec$ in conjunction with Lemma 5.8.

Now we show property $(\mathbf{3})$: The value returned by each read operation is the value written by the last preceding write operation according to $\prec$, or v_0 if there is no such write. Note that every version of the data object written in execution β is *uniquely* associated with a write operation in β. Lemma 5.8 implies that every version of the data object being written can be uniquely associated with *tag*. Therefore, to show that a read π returns the last preceding write, we only need to argue that the read returns the value associated with $T(\pi)$. From the write, read, and server protocols, it is clear that a value and/or its coded elements are always paired together with the corresponding tags at every state of every component of the system. In particular, the read returns the value from k coded elements by inverting the MDS code Φ; these k coded elements were obtained at some previous point by applying Φ to the value associated with $T(\pi)$. Therefore Definition 4.1 implies that the read returns the value associated with $T(\pi)$. □

E Proof of Liveness of CAS

Proof of Lemma 5.9. By examination of the algorithm we observe that termination of any operation depends on termination of its phases. So, to show liveness, we need to show that each phase of each operation terminates. Let us first examine the *query* phase of a read/write operation; note that termination of the *query* phase of a client is contingent on

receiving responses from a quorum. Every non-failed server responds to a *query* message with the highest locally available tag marked 'fin'. Since every server is initialized with $(t_0, v_0, \text{'fin'})$, every non-failed server has at least one tag associated with the label 'fin' and hence responds to the client's *query* message. Since the client receives responses from every non-failed server, property (**ii**) of Lemma 5.1 ensures that the *query* phase receives responses from at least one quorum, and hence terminates. We can similarly show that the *pre-write* phase and *finalize* phase of a writer terminate. In particular, termination of each of these phases is contingent on receiving responses from a quorum. Their termination is guaranteed from property (**ii**) of Lemma 5.1 in conjunction with the fact that every non-failed server responds, at some point, to a *pre-write* message and a *finalize* message from a write with an acknowledgment. Proof of termination of a reader's *finalize* phase is placed in Sec. 5.

, a writer propagated a *finalize* message with tag t. Let us denote by $Q_{pw}(t)$, the set of servers that responded to this write's *pre-write* phase. Now, we argue that all servers in $Q_{pw}(t) \cap Q_{fw}$ respond to the reader's *finalize* message with a coded element. To see this, let s be any server in $Q_{pw}(t) \cap Q_{fw}$. Since s is in $Q_{pw}(t)$, the server protocol for responding to a *pre-write* message implies that s has a coded element, w_s, at the point where it responds to that message. Since s is in Q_{fw}, it also responds to the reader's *finalize* message, and this happens at some point after it responds to the *pre-write* message. So it responds with its coded element w_s. From Lemma 5.1, it is clear that $|Q_{pw}(t) \cap Q_{fw}| \geq k$ implying that the reader receives at least k coded elements in its *finalize* phase and hence terminates. □

F Proof of correctness of CASGC

Proof of Theorem 6.2 (Sketch). We show atomicity and liveness in two steps. Note that, formally, CAS is an I/O automaton formed by composing the automata of all the nodes and communication channels in the system. In the first step, we construct a I/O automaton CAS$'$ which differs from CAS in that some of the actions of the servers in CAS$'$ are non-deterministic. However, from the perspective of the external responses of the client, CAS$'$ will be identical to CAS implying that CAS$'$ satisfies atomicity and liveness. In the second step, we will show that CASGC simulates CAS$'$. These two steps suffice to show that CAS$'$ satisfies both atomicity and liveness.

We now describe CAS$'$. The CAS$'$ automaton is identical to CAS with respect to the client actions, and to the server responses to *query* and *pre-write* messages and *finalize* messages from writers. A server's response to a *finalize* message from a read operation can be different in CAS$'$ as compared to CAS. In CAS$'$, if at the point of the receipt of the *finalize* message at the server, the read has already terminated, then, the server could respond either with the coded element, or not respond at all (even if it has the coded element)[xvi].

We now argue that CAS$'$ "simulates" CAS. Formally speaking, for every fair execution α' of CAS$'$, there is a natural corresponding execution α of CAS with an identical sequence of actions of all the components with one exception; when a server ignores a

[xvi] We note that, in the CAS$'$ automaton, the conditions on a server's response are dependent on the reader's state. In our algorithms, the server is unaware of the state of the reader because of the distributed nature of the system. Nonetheless, we will show that CASGC implements the CAS$'$ automaton under our system model.

read's *finalize* message in α', we assume that the corresponding message in α is indefinitely delayed. We now aim to show that α' satisfies liveness and atomicity by exploiting the fact that CAS satisfies these properties. For liveness, we require that all operations in α' terminate if the number of server failures is no bigger than f. Suppose that the number of server failures in α' is no bigger than f. To see that all operations in α' terminate, notice that the only difference between corresponding execution α of CAS, and a *fair* execution of CAS is that, in α, certain messages to a reader that pertain to a read operation that has already terminated are delayed indefinitely. However, since the read operation has already terminated, the delay of these messages does not affect termination of any operation in α. Because CAS satisfies the liveness condition, every operation in α terminates. Therefore, so does every operation in α'. It follows that CAS$'$ satisfies the liveness condition. Since CAS is atomic, α has atomic behavior, so α' does also. Therefore CAS$'$ satisfies both atomicity and liveness.

Now, we show that CASGC "simulates" CAS$'$. That is, for every execution α_{gc} of CASGC where the number of operations concurrent to a read operation is no bigger than δ, we construct a corresponding execution α' of CAS$'$ such that

- α' has the same external behavior (i.e., the same invocations, responses and failure events) as that of α_{gc}, and
- if α_{gc} is fair, then α' is fair as well.

Such a construction suffices to show that CASGC satisfies atomicity and liveness. To see this, note that because CAS$'$ satisfies atomicity, α' has atomic behavior. Since α_{gc} has the same external behavior as α', it has atomic behavior as well. If α_{gc} is a fair execution where the number of server node failures is at most f, then so is α'. Because CAS$'$ satisfies the liveness condition, every operation in α' terminates. Therefore, so does every operation in α_{gc}. Therefore CASGC satisfies the desired liveness condition.

We describe the execution α' step-by-step, that is, we consider a step of α_{gc} and describe the corresponding step of α'. We then show that our construction of α' has the properties listed above, i.e., we show that the external behavior is the same in both α' and α_{gc}; in addition, we show that if α_{gc} is fair, so is α'. Finally, we show that the execution α' that we have constructed is consistent with the CAS$'$ automaton.

We construct α' as follows. We first set the initial states of all the components of α' to be the same as they are in α_{gc}. At every step, the states of the client nodes and the message passing system in α' are the same as the states of the corresponding components in the corresponding step of α_{gc}. A server's responses on receipt of a message is the same in α' as that of the corresponding server's response in α_{gc}. In particular, we note that a server's external responses are the same in α_{gc} and α' even on receipt of a reader's *finalize* message, that is, if a server ignores a reader's finalize message in α_{gc}, it ignores the reader's finalize message in α' as well. The only difference between α_{gc} and α' is in the change to the server's internal state at a point of receipt of a *pre-write* message. At this point, the server performs garbage collection in α_{gc}, whereas it does not perform garbage collection in α'.

Note that the initial state, the server's response, and the client states at every step of α' are the same as the corresponding step of α_{gc}. Also note that a server that fails at a step of α_{gc} fails at the corresponding step of α' (even though the server states could be different in general because of the garbage collection). Hence, at every step, the external behavior of α' and α_{gc} are the same. This implies that the external behavior of the entire

execution α' is the same as the external behavior of α_{gc}. Furthermore, if α_{gc} is a fair execution, so is α'.

We now show that α' is consistent with the CAS$'$ automaton. Note that since the initial states of all the components are the same in the CAS$'$ and CASGC algorithms, the initial state of α' is consistent with the CAS$'$ automaton. We show that every step of α' is consistent with CAS$'$. It is easy to verify that at every step of α', the server internal states and client states are consistent with CAS$'$. The only step where it is non-trivial to show consistency with CAS$'$, is at a point of α' where a server ignores a *finalize* message from a reader. Consider such a point in α', and let t indicate the tag associated with the reader's finalize message. To be consistent with CAS$'$, we need to verify that, at this point α', the read operation that sent this message has already terminated. To see this, consider the corresponding point of α_{gc}. By construction, we know that the server ignores the reader's finalize message with tag t at this point in α_{gc}. From the server protocol in CASGC, we infer that at this point of α_{gc}, the server has $(t, \text{`null'}, \{*, \text{`gc'}\})$ in its list of stored triples. Lemma 6.3 implies that at this point of α_{gc}, the read operation has already terminated. Since, in our construction, the client behavior is identical in α_{gc} and α', the read operation has terminated at the point of α' in consideration as well. Therefore α' is consistent with the CAS$'$ automaton.

□